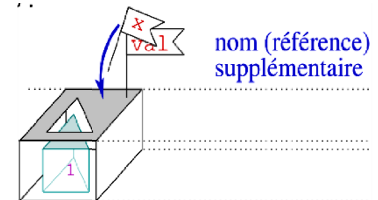
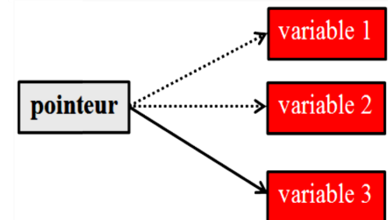
Chapitre VI : Les pointeurs en C++

VI Pointeurs et allocation dynamique [9]

Dans un programme, pour garder un lien vers une donnée (une variable), on utilise des références ou des pointeurs. En programmation, les pointeurs et références servent essentiellement à trois choses:



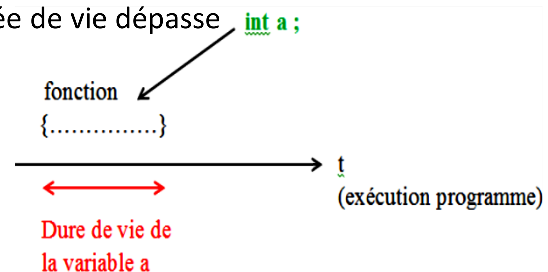
1. Permettre à plusieurs portions de code de partager des objets (données, fonctions,..) sans les dupliquer
   => **Référence**



2. Pouvoir choisir des éléments non connus a priori (au moment de la programmation)
   => **Généricité**

3. Pouvoir manipuler des objets dont la durée de vie dépasse la portée
   => **Allocation dynamique**

   La durée de vie de la variable a est égale au temps d'exécution de sa portée.



VI.2.1 Différents pointeurs et références

En C++, il existe plusieurs sortes de « pointeurs » (pointeur et références) :

- **Les références:** totalement gérées en interne par le compilateur. Très sûres, donc; mais sont fondamentalement différentes des vrais pointeurs.
- **Les « pointeurs intelligents » (smart pointers):** gérés par le programmeur, mais avec des gardes-fous. Il en existe 3: *unique_ptr*, *shared_ptr, weak_ptr* (avec #include <memory>)
- **Les « pointeurs « hérités de C » » (build-in pointers):** les plus puissants (peuvent tout faire) mais les plus « dangereux »

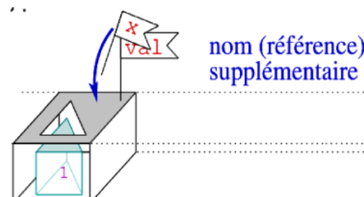
VI.2.1.1 Référence

Une référence est un autre nom pour un objet existant, un synonyme, un alias.
La syntaxe de déclaration d'une référence est:

<type>& nom_reference(identificateur) ;

Après une telle déclaration, **nom_reference** peut être utilisé partout où **identificateur** peut l'être.

```
int val(1) ;
int& x(val) ;
```



✍ C'est exactement ce que l'on utilise lors d'un passage par référence:

Exemple 1 :

```
#include <iostream>
using namespace std;
void f(int & a )
{ ++a ;  }
int main (void)
 {  int b=1;
    f(b) ;
    cout << " APRES b =  "<<b;
  return 0;
 }
```

```
 APRES b =   2
Process returned 0 (0x0)   execution time : 2.728 s
Press any key to continue.
```

Résultat de l'exécution

Exemple 2 : **Sémantique de const**

```
int i(3);
const int &   j(i);    // i et j sont les mêmes. On ne peut pas changer la valeur via j
// j=12 ;      // Erreur de compilation
i = 12 ;       // oui, et j aussi vaut 12
cout << " j =  "<<j;
```

```
j = 12
Process returned 0 (0x0)
Press any key to continue.
```

Résultat de l'exécution

✍ **Spécificités des références :** Contrairement aux pointeurs, une référence :

a. doit absolument être initialisée (vers un objet existant) :

```
int i;
int &  rj;    // Non, la r férence rj doit être liée à un objet !
```

b. ne peut être liée qu'à un seul objet :

```
int i;
int & ri;
int j(2) ;
ri = j ;   /* Ne veut pas dire que ri est maintenant un alias de j mais que
                                                   i prend la valeur de j */
j = 3 ;
cout<< i <<endl ; // affiche 2
```

c. ne peut pas être référencée

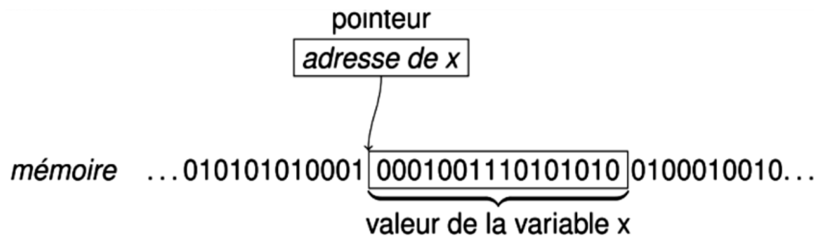
```
int i(3);
int &  ri(i);
int &  rri(ri); // Non
int &&  rri(ri); // Non plus !
```

```
Message
=== Build: Debug in test (compiler: GNU GCC Compiler) ===
In function 'int main()':
error: conflicting declaration 'int&& rri'
error: 'rri' has a previous declaration as 'int& rri'
warning: unused variable 'rri' [-Wunused-variable]
=== Build failed: 2 error(s), 1 warning(s) (0 minute(s), 0 second(s)) ===
```

⇒ On ne peut donc pas faire de tableau de références

VI.2. Pointeurs

Une variable est physiquement identifiée de façon unique par son **adresse**, c'est-à-dire l'adresse de l'emplacement mémoire qui contient sa valeur.
Un **pointeur** est une variable qui contient **l'adresse** d'un autre objet informatique (par ex, variable) => une « variable de variable »



Notes :

- Une référence n'est pas un «vrai pointeur» car ce n'est pas une variable en tant que telle.
- Une référence est «une autre étiquette».
- Un pointeur «une variable contenant une adresse»

✍ La déclaration d'un pointeur se fait selon la syntaxe suivante : **type*   identificateur ;** Cette instruction déclare une variable de nom **identificateur** de type pointeur sur une valeur de type **type**.

Exemple: déclare une variable **ptr** qui pointe sur une valeur de type int: **i t*** ptr ;

✍ L'initialisation d'un pointeur se fait selon la syntaxe suivante:

type* identificateur(adresse) ;

Exemple:

```
Int* ptr(NULL);
int *  ptr(&i);
int * ptr(new int(33));
```

nullptr : mot clé C++, spécifie que le pointeur ne pointe sur rien

- Pour le compilateur, une variable est un emplacement dans la mémoire,
- Cet emplacement est identifié par une adresse
- Chaque variable possède une et une seule adresse.
- Un pointeur permet d'accéder à une variable par son adresse.
- Les pointeurs sont des variables faites pour contenir des adresses.

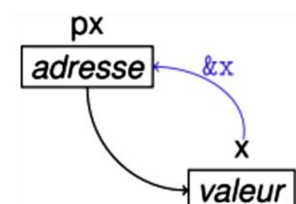
V.1. Opérateurs sur les pointeurs

C++ possède deux opérateurs particuliers en relation avec les pointeurs : **&** et *****.

- **&** est l'opérateur qui retourne l'adresse mémoire de la valeur d'une variable. Si x est de type **type**, &x est de type **type*** (pointeur sur type).

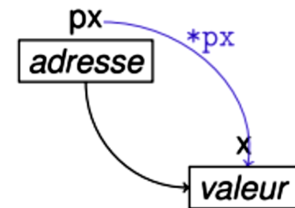
```
int  x(3);
int * px(NULL);
px = &x;
cout<< "Le pointeur px contient l'adresse"<<px<<endl ;
```

```
Le pointeur px contient l'adresse 0x28fee8
```



- ***** est l’opérateur qui retourne la valeur pointée par une variable pointeur. Si px est de type **type***, (*px) est la valeur de type **type** pointée par px.

```
int  x(3);
int * px(NULL);
px = &x;
cout<< "Le pointeur px contient l'adresse"<<*px<<endl ;
```




```
Le pointeur px pointe sur la valeur 3
```

Note : ***&i** est donc strictement équivalent à **i**

Attention aux confusions

- **int& id(i)** - id est une référence sur la variable entière i
- **&id** est l’adresse de la variable id

```
int i(3);
int& id(i);
cout<< " i = "<< i <<" id = "<< id <<endl ;
cout<< " Adresse de i = "<< &i  <<endl ;
```

```
i = 3 id = 3
adresse de i = 0x28fee8
```

- **int* id**; déclare une variable id comme un pointeur sur un entier
- ***id** (où id est un pointeur) représente le contenu de l’endroit pointé par id

```
int*  ptr(NULL);
int i = 2;
p = &i ;
cout<< " i = "<< i <<" id = "<< id <<endl ;
cout<< " Le pointeur p pointe sur la valeur  "<< *p <<endl ;
```

```
Le pointeur p pointe sur la valeur 2
```

VI.3. Pointeurs et Tableaux

Par convention, le nom d’une **variable** utilisé dans une partie droite d’expression donne le **contenu** de cette **variable** dans le cas d’une variable simple. Mais un **nom de tableau** donne l’**adresse du tableau** qui est l’adresse du premier élément du tableau.

Nous faisons les constatations suivantes :

- un **tableau** est une ***constante d’adressage*** ;
- un **pointeur** est une ***variable d’adressage***.

Ceci nous amène à regarder l’utilisation de pointeurs pour manipuler des tableaux, en prenant les variables: long i, tab[10], *pti ;

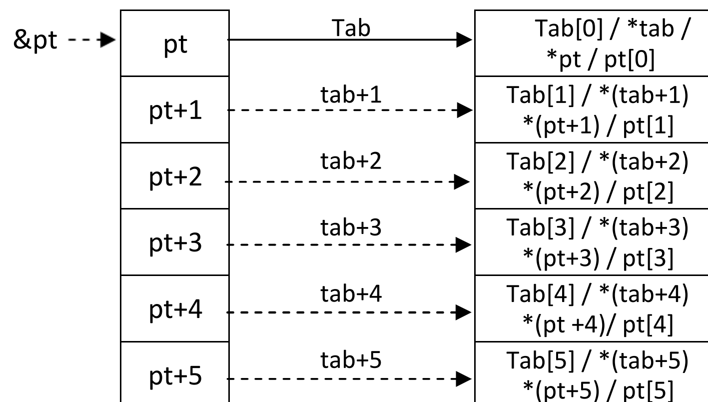
- tab est l’adresse du tableau (adresse du premier élément du tableau &tab[0] ;
- **pti = tab ;** initialise le pointeur pti avec l’adresse du début de tableau. Le & ne sert à rien dans le cas d’un tableau. pti = &tab est inutile et d’ailleurs non reconnu ou ignoré par certains compilateurs ;
- **&tab[1]** est l’adresse du 2e élément du tableau.
- **pti = &tab[1]** est équivalent à: pti = tab ; où pti pointe sur le 1er élément du tableau.

- **pti += 1 ;** fait avancer, le pointeur d'une case ce qui fait qu'il contient l'adresse du $2^{ème}$ élément du tableau.

Nous pouvons déduire de cette arithmétique de pointeur que :

- **tab[i]** est équivalent à ***(tab +i)**.
- ***(pti+i)** est équivalent à **pti[i]**.

La figure suivante est un exemple dans lequel sont décrites les différentes façons d'accéder aux éléments d'un tableau **tab** et du pointeur **pt** après la définition suivante:
int tab[6], ***pt = tab;

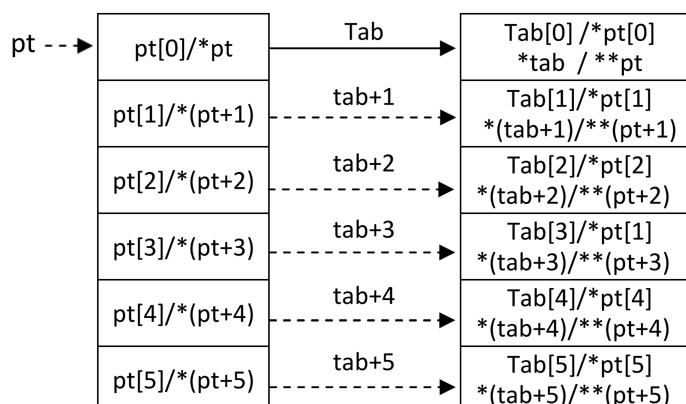


VI.4. Tableau de pointeurs

La définition d'un tableau de pointeurs se fait par : **type** ***nom[taille];

Le premier cas d'utilisation d'un tel tableau est celui où les éléments du tableau de pointeurs contiennent les adresses des éléments du tableau de variables. La figure suivante illustre un exemple d'utilisation de tableau de pointeurs à partir des définitions de variables suivantes :

int tab[3], ***pt[6] = {tab,tab+1,tab+2,tab+3,tab+4,tab+5};



VI.5. Allocation dynamique

Il y a trois façons d'allouer de la mémoire en C++

1. **Allocation statique :** La réservation mémoire est déterminée à la **compilation.** L'espace alloué statiquement est déjà réservé dans le fichier exécutable du programme lorsque le système d'exploitation charge le programme en mémoire pour l'exécuter. L'avantage de l'allocation statique se situe essentiellement au niveau des performances, puisqu'on évite les coûts de l'allocation dynamique à l'exécution.

2. **Allocation dynamique:** La mémoire est réservée pendant **l'exécution** du programme. L'espace alloué dynamiquement ne se trouve pas dans le fichier exécutable du programme lorsque le système d'exploitation charge le programme en mémoire pour l'exécute.
   Par exemple, les tableaux de taille variable (**vector**), les chaînes de caractères de type **string**.

Dans le cas **particulier des pointeurs**, l'allocation dynamique permet également de réserver de la mémoire **indépendamment de toute variable**: on pointe directement sur une zone mémoire plutôt que sur une variable existante.

VI.5.1. Allocation d'une case mémoire

C++ possède deux opérateurs **new** et **delete** permettant **d'allouer** et de **libérer dynamiquement de la mémoire**.

Syntaxe: pointeur = **new** type ⇒ Réserve une zone mémoire de type **type** et affecte l'adresse dans la variable **pointeur.**

Exemple :

```
int* p;
p = new int ;
*p = 2 ;
cout<< " p = "<< p <<" *p = "<< *p <<endl ;
```

```
p = 0x350cf0 *p = 2
Process returned 0 (0x0)
Press any key to continue.
```

```
int* p;
p = new int (2);
cout<< " p = "<< p <<" *p = "<< *p <<endl ;
```

VI.5.2 Libérer la mémoire allouée

Syntaxe : delete pointeur ⇒ libère la zone mémoire allouée au pointeur

C'est-à-dire que cette zone mémoire peut maintenant être utilisée pour autre chose. **On ne peut plus y accéder !**

✍ **Bonnes pratiques**

- Faire suivre tous les **delete** de l'instruction **« pointeur = NULL; »**.
- Toute zone mémoire allouée par un **new** doit impérativement être libérée par un **delete** correspondant !